Chapter 2

Literature Review

Chapter 2 Literature Review

The literature review conducted covers the following topics:

i. software quality

ii. software quality models

iii. software metrics

iv. software metrics for requirements analysis

v. requirements analysis

vi. software tools

These topics are discussed in the following sections.

2.1 Software Quality

Pressman [1992] states that software quality is defined as conformance to explicitly stated functional and performance requirements, explicitly documented development standards and implicit characteristics that are expected of all professionally developed software.

Garvin states that software quality has five definitions, namely transcendent definition, user-based definition, product-based definition, manufacturing-based definition and value-based definition [cited by Vilet, 1993]. The transcendent definition of quality concerns innate excellence. It is the type of quality assessment that is usually applied to novels. Likewise, the software engineering expert may have developed a good feeling

Table 2-2 Metric of understandability

1.1	Are variable names descriptive of the physical or functional property represented?
1.2	Is vestigial code adequately explained?
1.3	Are inputs, outputs and assumptions of program modules clearly stated via parameter lists or comment cards?
1.4	Does each program module (where module is defined to be a function / subroutine / statement function or piece of code entered from ASSIGN, GOTO pairs) contain a header block of commentary which describes program name, effective date, accuracy requirements, purpose, limitations and restrictions, modification history, inputs and outputs, method, assumptions and error recovery procedures for all foreseeable error exits that may exist?
1.5	Do uniquely recognizable functions contain adequate descriptive information (e.g. comments) so that the purpose of each is clear?
1.6	Are decision points and subsequent branching alternatives adequately described?
1.7	Are adequate descriptions provided to allow correlation of variable names with the physical property or entity which they represent?
1.8	Are all unusual termination conditions described?
1.9	Are procedures for recovery from errors included and described?
1.10	Does the program contain single, separate modules which have been referenced (called) from several places as opposed to redundant blocks or code?
1.11	Are cross-reference listings of variable names supplied? Is a map (figure, chart, table, etc.) of calling and called subroutines supplied?
1.12	Are indentations, blank lines, lines or boxes of X's or *'s used to separate pieces of code?
1.13	Are deviations from forward (i.e. downward) logical flow adequately described?
1.14	Has the ordering: commentary header block, specification statements, then executable code been followed?
1.15	Is there a transfer to all labeled statements and is the transfer easily located?
1.16	Are all elements of an array functionally related?
1.17	Have parentheses or some other technique been used to eliminate ambiguity in the order to mode (real, integer, single precision, double precision) of evaluation of arithmetic expression?
1.18	Is there, at most, one parameter value assignment per line of code?
1.19	Is there, at most, one executable statement per line of code?
1.20	Are all subscripts included in all uses of dimensioned arrays?
1.21	Does associated descriptive commentary have a low "Fog Index"?
1.22	Are programs or program elements which use this program or element identified?
1.23	Are programs or program elements which are used by this program or element identified?

2.2.1 Boehm's Software Quality Model

Boehm, Brown, Kaspar, Lipow, MacLeod and Merrit's study [Boehm et al, 1978] concentrates on quantitative measures for FORTRAN source code. They identify a set of quality software characteristics. For each characteristic, they define a metric that provides a quantitative measure of the degree to which a FORTRAN program has the associated characteristic. The overall quality of the program can be defined as some function of the metric values.

The eleven software characteristics identified are understandability, completeness, conciseness, portability, consistency, maintainability, testability, usability, reliability, structuredness and efficiency [Boehm et al, 1978]. This original set of characteristics was found to be highly overlapping. Therefore, this set of characteristics was revised. The revised set of characteristics forms into a hierarchy of characteristics. The higher level relates more to user needs while lower level relates more to specific metrics. They are also much more satisfactory with respect to being mutually exclusive and exhaustive. This hierarchy of characteristics is shown in Figure 2-1 [Boehm et al, 1978].

The hierarchy of characteristics indicates that the higher level general utility includes portability, as-is utility and maintainability. As-is utility requires a program to be reliable, adequately efficient and human-engineered. It does not, however, require the user to test the program, understand its internal working, modify it, or try to use it elsewhere. Maintainability requires the user to be able to understand, modify and test the program. Maintainability is also aided by good human-engineering. It does not, however, depend on the program's reliability, efficiency or portability. The lower level

provides a set of primitive characteristics. These primitive characteristics are strongly different with respect to each other. They combine into sets of necessary and sufficient conditions for the intermediate level characteristics [Boehm et al, 1978]. The descriptions on these software characteristics are shown in Table 2-1 [Boehm et al, 1978]. An example of the software characteristic metric: understandability is shown in Table 2-2 [Boehm et al, 1978].

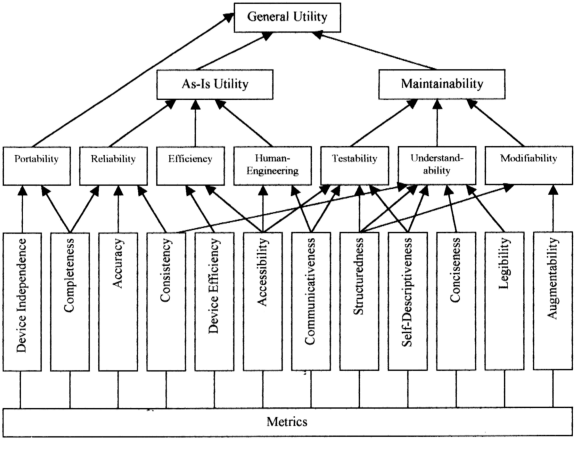

Figure 2-1 Boehm's software quality model

Table 2-1 Descriptions on software characteristics

No.	Software Characteristic	Description
1	Understandability	A software product possesses the characteristic understandability to the extent that the purpose of the product is clear to the evaluator.
2	Completeness	A software product possesses the characteristic completeness to the extent that all of its parts are present and each of its parts is fully developed.
3	Conciseness	A software product possesses the characteristic conciseness to the extent that no excessive information is present.
4	Portability	A software product possesses the characteristic portability to the extent that it can be operated easily and well on computer configurations other than its current one.
5	Consistency	A software product possesses the characteristic consistency to the extent that it contains uniform notation, terminology and symbology within itself.
6	Maintainability	A software product possesses the characteristic maintainability to the extent that it facilitates updating to satisfy new requirements.
7	Testability	A software product possesses the characteristic testability to the extent that it facilitates the establishment of acceptance criteria and supports evaluation of its performance.
8	Usability	A software product possesses the characteristic usability to the extent that it is convenient and practical to use.
9	Reliability	A software product possesses the characteristic reliability to the extent that it can be expected to perform its intended functions satisfactorily.
10	Structuredness	A software product possesses the characteristic structuredness to the extent that it possesses a definite pattern of organization of its interdependent parts.
11	Efficiency	A software product possesses the characteristic efficiency to the extent that it fulfills its purpose without waste of resources.
12	Device-Independence	A software product possesses the characteristic device-independence to the extent that it can be executed on computer hardware configurations other than its current one.
13	Accuracy	A software product possesses the characteristic accuracy to the extent that its outputs are sufficiently precise to satisfy their intended use.
14	Accessibility	A software product possesses the characteristic accessibility to the extent that it facilitates the selective use of its components.

15	Communicativeness	A software product possesses the characteristic communicativeness to the extent that it facilitates the specification of inputs and provides outputs whose form and content are easy to assimilate and useful.
16	Legibility	A software product possesses the characteristic legibility to the extent that its functions and those of its component statements are easily discerned by reading the code.
17	Self-Descriptiveness	A software product possesses the characteristic self-descriptiveness to the extent that it contains enough information for a reader to determine its objectives, assumptions, constraints, inputs, outputs, components and status.
18	Augmentability	A software product possesses the characteristic augmentability to the extent that it easily accommodates expansions in data storage requirements or component computational functions.
19	Human-Engineering	A software product possesses the characteristic human engineering to the extent that it fulfills its purpose without wasting user's time and energy or degrading their morale.
20	Modifiability	A software product possesses the characteristic modifiability to the extent that it facilitates the incorporation of changes once the nature of the desired change has been determined.

Table 2-2 Metric of understandability

1.1	Are variable names descriptive of the physical or functional property represented?
1.2	Is vestigial code adequately explained?
1.3	Are inputs, outputs and assumptions of program modules clearly stated via parameter lists or comment cards?
1.4	Does each program module (where module is defined to be a function / subroutine / statement function or piece of code entered from ASSIGN, GOTO pairs) contain a header block of commentary which describes program name, effective date, accuracy requirements, purpose, limitations and restrictions, modification history, inputs and outputs, method, assumptions and error recovery procedures for all foreseeable error exits that may exist?
1.5	Do uniquely recognizable functions contain adequate descriptive information (e.g. comments) so that the purpose of each is clear?
1.6	Are decision points and subsequent branching alternatives adequately described?
1.7	Are adequate descriptions provided to allow correlation of variable names with the physical property or entity which they represent?
1.8	Are all unusual termination conditions described?
1.9	Are procedures for recovery from errors included and described?
1.10	Does the program contain single, separate modules which have been referenced (called) from several places as opposed to redundant blocks or code?
1.11	Are cross-reference listings of variable names supplied? Is a map (figure, chart, table, etc.) of calling and called subroutines supplied?
1.12	Are indentations, blank lines, lines or boxes of X's or *'s used to separate pieces of code?
1.13	Are deviations from forward (i.e. downward) logical flow adequately described?
1.14	Has the ordering: commentary header block, specification statements, then executable code been followed?
1.15	Is there a transfer to all labeled statements and is the transfer easily located?
1.16	Are all elements of an array functionally related?
1.17	Have parentheses or some other technique been used to eliminate ambiguity in the order to mode (real, integer, single precision, double precision) of evaluation of arithmetic expression?
1.18	Is there, at most, one parameter value assignment per line of code?
1.19	Is there, at most, one executable statement per line of code?
1.20	Are all subscripts included in all uses of dimensioned arrays?
1.21	Does associated descriptive commentary have a low "Fog Index"?
1.22	Are programs or program elements which use this program or element identified?
1.23	Are programs or program elements which are used by this program or element identified?

2.2.2 McCall's Factor Criteria Metric Model

The earliest approach to software measurement that can reasonably be termed a method is from the work of Boehm, Brown, Kaspar, Lipow, MacLeod and Merrit on software quality [Shepperd, 1995]. McCall, Richards and Walters codified this work into the Factor Criteria Metric (FCM) method. The principal aim of the FCM model is to obtain a quantitative view of quality. The FCM model was originally developed for the United States Air Force. Its use is promoted within the United States Department of Defense (DoD) to evaluate the quality of software product [Fenton et al, 1997]. The FCM model is highly influential because it forms part of the Institute of Electrical and Electronic Engineers (IEEE) software quality metrics methodology standard [Institute of Electrical and Electronic Engineers Inc, 1989]. Besides, the FCM model has been used in formulating the ISO 9126 Criteria for Evaluating Software [Fenton et al, 1997].

McCall, Richards and Walters distinguish two levels of quality attributes. The higher level quality attributes are termed as quality factors. The quality factors cannot be measured directly. The second level of quality attributes are termed as quality criteria. The quality criteria can be measured. Finally, the measure for the extent to which a quality factor is satisfied can be obtained by combining the ratings for the individual quality criterion for that particular quality factor [Vilet, 1993]. The McCall's Factor Criteria Metric (FCM) model is shown in Figure 2-2 [Fenton et al, 1997]. The descriptions on the quality factors and quality criteria are given in Table 2-3 and Table 2-4 respectively [Vilet, 1993].

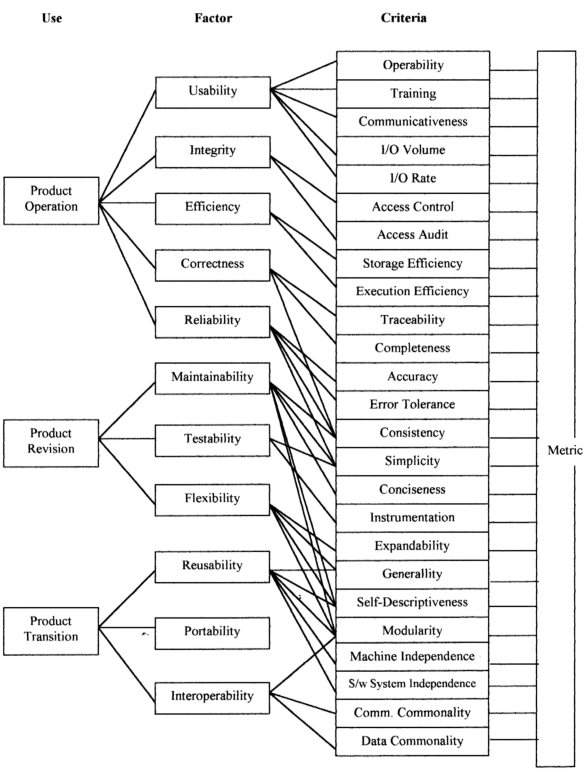

Figure 2-2 McCall's Factor Criteria Metric model

Table 2-3 Descriptions on quality factors

No.	Quality Factor	Description
1	Usability	Effort required to learn, operate, prepare input and interpret output of a program.
2	Integrity	Extent to which access to software or data by unauthorised persons can be controlled.
3	Efficiency	Amount of computing resources and code required by a program to perform a function.
4	Correctness	Extent to which a program satisfies its specifications and fulfills the user's mission objectives.
5	Reliability	Extent to which a program can be expected to perform its intended function with required precision.
6	Maintainability	Effort required to locate and fix an error in an operational program.
7	Testability	Effort required to test a program to ensure that it performs its intended function.
8	Flexibility	Effort required to modify an operational program.
9	Reusability	Extent to which a program or parts of the program can be reused in other applications.
10	Portability	Effort required to transfer a program from one hardware and/or software environment to another.
11	Interoperability	Effort required to couple one system with another.

Table 2-4 Descriptions on quality criteria

No.	Quality Criterion	Description
1	Access audit	Ease with which software and data can be checked for compliance with standards or other requirements.
2	Access control	Provisions for control and protection of software and data.
3	Accuracy	Precision of computations and output.
4	Communication commonality	Degree to which standard protocols and interfaces are used.
5	Completeness	Degree to which a full implementation of required functionality has been achieved.
6	Communicativeness	Ease with which inputs and outputs can be assimilated.
7	Conciseness	Compactness of source code in terms of lines of code.
8	Consistency	Use of uniform design and implementation techniques and notations throughout a project.
9	Data commonality	Use of standard data representations.
10	Error tolerance	Degree to which continuity of operation is ensured under adverse conditions.
11	Execution efficiency	Run-time efficiency of the software.
12	Expandability	Degree to which storage requirements or software functions can be expanded.
13	Generality	Breadth of potential application of software components.
14	Hardware independence	Degree to which software is independent on underlying hardware.
15	Instrumentation	Degree to which software provides for measurements of its use or identification of errors.
16	Modularity	Provision of highly independent modules.
17	Operability	Ease of operation of software.
18	Self-documentation	Provision of in-line documentation that explains implementation of components.
19	Simplicity	Ease with which software can be understood (this usually implies avoidance of practices which increase complexity of software).
20	Software system independence	Degree to which software is independent of its software environment.
21	Storage efficiency	Run-time storage requirements of software.
22	Traceability	Ability to link software components to requirements.
23	Training	Ease with which new users can be made to use the system.

McCall's FCM model has 41 metrics to measure 23 quality criteria generated from the quality factors. Measuring a factor requires the consideration of the checklists of conditions that applies to the requirements (R), design (D) and/or implementation (I). Each condition in a checklist is designated as 'yes' or 'no' depending on whether it is met.

The following shows how to measure a quality factor. An example of the quality factor, correctness is explained below. The quality factor, correctness has three quality criteria, namely completeness, traceability and consistency. The checklist for completeness is shown in Table 2-5 [Fenton et al, 1997].

Table 2-5 Checklist for quality criterion completeness

No	Condition	Phase
1	Unambiguous references.	R,D,I
2	All data references defined, computed or obtained from external source.	R,D,I
3	All defined functions used.	R,D,I
4	All referenced functions defined.	R,D,I
5	All conditions and processing defined for each decision point.	R,D,I
6	All defined and referenced calling sequence parameters agree.	D,I
7	All problem reports resolved.	R,D,I
8	Design agrees with requirements.	D
9	Code agrees with design.	I

Keys: R Requirement D Design I Implementation

From Table 2-5, there are six conditions that apply to requirement (R), eight conditions that apply to design (D) and eight conditions that apply to implementation (I). Hence, the measurement for completeness is:

$$\frac{(\text{No. of yes for R/6}) + (\text{No. of yes for D/8}) + (\text{No. of yes for I/8})}{3}$$

Traceability measure and consistency measure are both obtained in the similar manner. Finally, the quality factor, correctness measure, is the mean of completeness measure, traceability measure and consistency measure. In this example, all the criteria of the factor and all the conditions in each checklist of the criteria are given the same weight. It is, however, possible to use different weights so that each weight reflects the importance of a criterion to the factor or importance of a condition in a checklist to a criterion.

2.3 Software Metrics

2.3.1 Definition

Pressman [1992] defines that generally software metrics refer to a broad range of measures for software. Fenton and Pfleeger [1997] define measurement as the process by which numbers or symbols are assigned to attributes of entities in the real world according to clearly defined rules. An entity is an object or an event in the real world. For example, a car and a journey. An attribute is a feature or property of an entity. The attributes are important in distinguishing one entity from another. For example, colour of a car and cost of a journey. Colour is the attribute of the entity car. Cost is the attribute of the entity journey. Shepperd [1995] defines measurement as the process of assigning symbols, usually numbers, to represent an attributes of the entity of interest by rule. Boehm, Brown, Kaspar, Lipow, MacLeod and Merrit [1978] define the term metric as a measure of the extent to which a product possesses and exhibits a certain quality characteristic.

2.3.2 Purpose

According to Pressman [1992], software is measured for the following purposes:

i. to indicate the quality of the product

ii. to assess the productivity of those involved in producing the product

iii. to assess the benefits derived from the new methods and tools

iv. to form a baseline for estimation

v. to help justify requests for new tools or additional training

2.3.3 Measurement Scale

There are five major types of measurement scales, namely, nominal scale, ordinal scale, interval scale, ratio scale and absolute scale. Each measurement scale carries more information than the one before. The nominal scale involves classification by sorting elements into classes with respect to a certain attribute. The ordinal scale enables classes to be ordered. The interval scale captures information about the size of the intervals that separate the classes. The ratio scale provides empirical relations to capture ratios. The absolute scale enables only one measurement for an element [Fenton et al, 1997].

2.4 Software Metrics for Requirements Analysis

The metrics that can be used during the requirements analysis include the following [Jalote, 1991]:

i. function points that predicts the size of a system

ii. number of errors found in a requirements specification that can be used to assess the quality of the software requirements specification

iii. frequency of requirements change that can be used to assess the stability of requirements

These metrics are explained in the following sections.

2.4.1 Function Points

Function point analysis (FPA) was developed by Allan Albrecht, a software expert working in the late 1970s at IBM. The basic principle was to focus upon the size of the requirements specification or system functionality [Albrecht, 1979].

FPA begins with analysing the requirements specification to classify the requirements to five basic function types. These five function types are external input types (e.g. file names), external output types (e.g. reports), enquiries (interactive input needing a response), external files (files shared with other software systems) and internal files (invisible outside system). The next step is to classify the functions according to their complexity. The complexity levels are simple, average and complex. The complexity level of each function is determined by the following determinants: number of file access, number of record type access and number of data element involved in the

function. Each function type is given a weight for each of the three complexity levels. Therefore, once the function type and complexity level of each function have been obtained, the weighted count for the system can be calculated. This count is known as unadjusted function count (UFC). For example, a system comprising three simple external input, one complex external input and two average external output scores 25 UFC [Shepperd, 1995].

Albrecht [1979] also considered that there might be other factors that could influence the 'size' of the functional requirements for a system. These factors are encapsulated into fourteen general system characteristics (GSCs) that contribute to an overall value adjustment factor (VAF) for a system. Each characteristic is rated in terms of its degree of influence (DI) on the system on a scale of 0 (not applicable) to 5 (essential). The DIs for all the fourteen characteristics are added to derive a total degree of influence (TDI). Therefore, the TDIs may range from 0 to 70 (70 is obtained if the degree of influence of all the fourteen GSCs is 5 or essential). The VAF is calculated as VAF = 0.65 + (0.01 * TDI). Hence, VAF for a system may range from 0.65 to 1.35. The VAF value is then used to modify the unadjusted function count (UFC) to obtain the function point (FP) for the system as UFC * VAF.

The FP of a system can be turned into productivity measure. For example, the FP of a system can be divided with the number of persons to obtain the number of functions that a person should be assigned to. The FP of a system can also be divided with the number of months to obtain the number of functions that needs to be completed in a month [Shepperd, 1995].

2.4.2 Number of Errors Found

This metric can be useful to assess the quality of the requirements specification. The errors can be categorised into different types that include the number of missing functional requirements, number of ambiguity in the requirements specification, number of unverifiable requirements and number of incorrect requirements. The number of errors can be obtained from the requirements review report. The error data can be weighted to produce a single quality number. However, no such effort has been done [Jalote, 1991].

2.4.3 Change Request Frequency

The change request frequency can be used to assess the requirements stability and to predict the amount of requirements change in later stages [Jalote, 1991]. The change requests may come from clients. For example, a client may request for a new functionality or request for a report in a different format. The change requests may also come from the developers. For example, a developer may request to change the requirements due to difficulty in implementation.

Many organisations have formal methods to request and incorporate the requirements changes. Therefore, change data can be obtained from these formal change approval procedures. The change request frequency can be plotted against time. For most projects, the change request frequency decreases with time. If the frequency does not decrease with time, it can mean that the requirements analysis has not been done properly. The change request frequency can also be used to freeze requirements when the frequency goes below an acceptable threshold determined based on experience and historical data [Jalote, 1991].

2.5 Requirements Analysis

Each proposed model of the software development process includes activities aimed at gathering requirements that is understanding of what the customers and users expect the system to do. The requirements analysis phase involves requirements elicitation and requirements specification [Pfleeger, 1998]. According to Pfleeger [1998], a requirement is a feature of the system or a description of something the system is capable of doing in order to fulfil the purpose of the system.

2.5.1 Requirements Elicitation

The requirements analysis begins with problem analysis to understand the problem. Problem analysis involves working with customers to elicit requirements with the following methods [Schach, 1997; Pfleeger, 1998; Jalote, 1991; Vilet, 1993]:

i. asking questions through interviews, sending questionnaires to relevant members of the client organisation
ii. observing working processes, using scenarios
iii. developing prototypes of whole or part of the proposed system
iv. reading existing documents
v. examining various forms used
vi. demonstrating similar systems or derivation from an existing system

The information collected is in the form of answers to questions and questionnaires, and information from documentation. It is also necessary to organise the information and resolve contradictions in the information gathered from different parties [Jalote, 1991].

2.5.1.1 Conducting Requirements Gathering Interviews

The client usually sets up the initial interviews. Additional interviews may be scheduled during the interview process. Interviews continue until the requirements gathering team is convinced that it has elicited all the relevant information from the client and future users of the product [Jalote, 1991].

There are two basic types of interview, namely structured and unstructured interview [Curzon, 1995]. In a structured interview, specific preplanned close-ended questions are posed. In an unstructured interview, open-ended questions are asked to encourage the interviewee to speak out.

According to Mason and Willcocks [1994], the interviewer should ensure the following before the interview:

i. choose the right person to interview

ii. choose an appropriate venue

iii. make an appointment with the interviewee and prepare the interview

The interviewer should ensure the following during the interview:

i. observe good manners

ii. use appropriate language and style

iii. do not interrupt when the interviewee is talking

iv. avoid yes no question

v. do not express own opinion

vi. try to distinguish between fact and opinion

vii. distinguish need and desire

Curzon's guidelines to manage interviews to hire staff [1995] can be modified to suit requirements gathering interviews (RGIs). According to Curzon [1995], guidelines to prepare the interview include the following:

i. screen the candidates

ii. determine the number of candidates to interview

iii. decide upon the length of the interview

iv. set the interview date

v. create the proper physical setting

vi. decide on whether to tape the interview

vii. notify the interviewee

viii. confirm the interview the day before conducting the interview

The guidelines to conduct the interview include the following [Curzon, 1995]:

i. relax and concentrate

ii. open the interview properly

iii. ask the already prepared questions

iv. take notes

v. watch both tone and body language

vi. give the candidafe time to respond

vii. be careful with responses

viii. let the interviewee talk

ix. do not be nervous

x. repeat questions if necessary

xi. be fair to people with disabilities

xii. close the interview properly

The guidelines to prepare questions for an interview are similar to the guidelines to design questionnaires explained in Section 2.5.1.2.

After the interview, the interviewer should prepare a written report outlining the results of the interview. People who were interviewed should be given a copy of the report so that they can clarify statements or add overlooked items [Schach, 1997].

2.5.1.2 Designing Questionnaires

The requirements gathering team can also send questionnaires to the relevant members of the client organisation to elicit needs. This method is appropriate when the opinions of many individuals need to be obtained. Besides, a carefully thought-out written answer may be more accurate than an immediate verbal response to a question posed by an interviewer. Unlike interview, however, there is no way that a question can be posed in response to an answer because questionnaires are preplanned [Schach, 1997].

The following are the guidelines to design a questionnaire [Converse et al, 1986]:

i. Use simple language to ensure questionnaire simplicity

The questions for the questionnaires should be expressed with common words that come easily to the tongue of educated people. Short questions should be used when possible. Double negative questions should not be used. For example, "Students should not be allowed to use the new computer lab." Agree or disagree. Potentially ambiguous words should be clarified to indicate their scope in the questions. For example, the term neighbourhood can indicate houses in the same road or houses located within three roads in every direction.

ii. Make questions conceptually clear

This can be the most difficult task for the questionnaire designers. This is because they usually think and are trained in the technical manner that others may not understand. For example, mathematical abstractions are difficult for the public. Questions on variances or standard deviations should not be asked.

iii. Ensure that questions asked are manageable

The questions asked should be on personal facts rather than on opinions. The examples of personal facts include a person's experience and a person's behaviour. Hypothetical questions should not be asked unless it is really necessary. Hypothetical questions include questions that require people to imagine what might happen or what they might do if things had been otherwise. Questions should focus on the current, the specific and the real. This is because memory questions involving recall of the past are difficult to answer. Therefore, answers obtained from memory questions may not indicate the real situation. Memory questions can be questions on events that were so trivial that people have hardly given them a second thought since. Memory questions can also be questions on events that happened long ago or questions that require recall of many separate events.

iv. Questions asked should be specific and not general

For example, "how are you now" is a general question, while "how is your job now" is a specific question. People may answer in terms of their health, job or marriage for the general question. Therefore, answers of a general question may not be easy to analyse.

2.5.1.3 Observing

A newer way of eliciting needs is to set up video cameras within the workplace to record exactly what is being done. This method is inappropriate if the employees view the cameras as an invasion of privacy. If people feel threatened or uncomfortable, it can be difficult to obtain the necessary information [Schach, 1997].

2.5.1.4 Using Scenarios

Scenario is another technique to elicit needs. A scenario can be depicted in a number of ways. One technique is the use of a storyboard showing a series of diagrams depicting the sequence of events. A tree can also represent scenarios. The nodes of the tree correspond to specific screens while the branches represent the possible actions that the user can take at that point in the scenario [Schach, 1997].

2.5.2 Requirements Specification

After requirements are elicited, requirements need to be captured in the requirements documents that can be rewritten in a more mathematical representation [Pfleeger, 1998]. The requirements specification techniques include using natural languages, pictures and formal languages [Vilet, 1993]. Later, requirements should be verified to ensure that they are complete, correct and consistent [Pfleeger, 1998].

The final product of the requirements analysis is a software requirements specification (SRS). Table 2-6 shows the structure of the SRS following the Institute of Electrical and Electronic Engineers (IEEE) Guide to Software Requirements Specifications [cited by Vilet, 1993].

Table 2-6 Structure of SRS following IEEE Guide to SRSs

1	**Introduction**
1.1	Purpose
1.2	Scope
1.3	Definitions, Acronyms and Abbreviations
1.4	References
1.5	Overview
2	**General Description**
2.1	Product Perspective
2.2	Product Functions
2.3	User Characteristics
2.4	General Constraints
2.5	Assumptions and Dependencies
3	**Specific Requirements**
3.1	Functional Requirements
3.1.1	Functional Requirement 1
3.1.1.1	Introduction
3.1.1.2	Inputs
3.1.1.3	Processing
3.1.1.4	Outputs
3.1.2	Functional Requirement 2
...	...
3.1.n	Functional Requirement n
3.2	External Interface Requirements
3.2.1	User Interfaces
3.2.2	Hardware Interfaces
3.2.3	Software Interfaces
3.2.4	Communications Interfaces
3.3	Performance Requirements
3.4	Design Constraints
3.4.1	Standards Compliance
3.4.2	Hardware Limitations
...	...
3.5	System Attributes
3.5.1	Security
3.5.2	Maintainability
...	...
3.6	Other Requirements
3.6.1	Database
3.6.2	Operations
3.6.3	Site Adaptation

An SRS has three main parts, namely Introduction, General Description and Specific Requirements.

The Introduction includes the Purpose, Scope, Definition, Acronyms and Abbreviations, References and Overview section. The Purpose section describes the main objectives of the SRS, key people involved in the project and their contributions and responsibilities. The Scope section identifies the product to be developed stating its name, functions and distinct features. The Definitions, Acronyms and Abbreviations section has lists of definitions, acronyms and abbreviations that serve as a glossary. The References section specifies the internal and external references for the SRS. The Overview section describes briefly on the contents of the SRS [cited by Vilet, 1993].

The Overall Description includes the Product Perspective, Product Functions, User Characteristics, General Constraints and Assumptions and Dependencies section. The Product Perspective section describes the placement of the product in a proper perspective within the overall system. If the product is an independent product, this section states its relationship with other independent products. If the product is part of a larger product, this section identifies the interface with other products and evaluates the extra loads introduced by integrating the product into the overall product. The Product Functions section describes briefly the functions to be performed by the product. The User Characteristics section describes the user behaviour characteristics. The General Constraints Section identifies the hardware limitations and government policies. The Assumptions and Dependencies section states assumptions and dependencies. For example, the product works only with a certain operating system [cited by Vilet, 1993].

The Specific Requirements contains all the requirements of the product. The Specific Requirements has Functional Requirements, External Interface Requirements, Performance Requirements, Design Constraints, System Attributes and Other Requirements section. The Functional Requirements section defines the fundamental actions that must take place in the product in receiving and processing the input and in processing and generating the output. These actions are generally listed as "shall" statements starting with "The system shall...". The External Interface Requirements section specifies the detailed description on all the inputs into the product and all the outputs from the product. The external interfaces of the product include the system interfaces, user interfaces, hardware interfaces, software interfaces and communications interfaces. The Performance Requirements section states the static and dynamic numerical requirements placed on the product or on the human interaction with the product. An example of the static numerical requirement is the number of simultaneous users to be supported. An example of the dynamic numerical requirement is the number of transactions to be processed within a certain period of time. The Design Constraints section specifies the constraints imposed by other standards or by hardware limitations. The System Attributes section states the attributes of the product such as reliability, availability, security, maintainability and portability that serve as requirements. These requirements can state the following [cited by Vilet, 1993]:

i. factors required to establish the required level of reliability of the product at the time of delivery (reliability)

ii. factors required to guarantee a defined level of availability of the product (availability)

iii. factors required to protect the product from accidental or malicious access, use, modification, destruction or disclosure (security)

iv. factors required to ease the maintenance of the product (maintainability)

v. factors required to ease the porting of the product to other operating systems (portability)

The Other Requirements section specifies the logical database requirements, operations requirements and site adaptation requirements.

2.6 Software Tools

Some software tools for the requirements analysis and some measurement tools are tabulated in Table 2-7 and Table 2-8, respectively [Department of Computing and Information Science Queen's University at Kingston, 2000].

Table 2-7 Software tools for requirements analysis

Tool	Vendor
Analyst Pro	Goda Software Inc.
RDD-100	Ascent Logic Corporation
RDT	Igatech Systems Pty. Ltd.
XTie-RT	Teledyne Brown Engineering

Table 2-8 Measurement tools

Tool	Vendor	Description
Cantata	Information Processing Ltd (IPL)	metrics for test coverage of C programs
Cantata++	Information Processing Ltd (IPL)	metrics for test coverage of C++ programs
Checkpoint	Software Productivity Research Inc.	estimation and measurement
Costar	Softstar Systems	COCOMO cost estimation
ESTIMATE Professional	Software Productivity Research Inc.	software project planning and estimation
Resource Standard Metrics	M Squared Technologies	quality analysis and metrics for C, C++ and Java programs

The attempt to locate a tool that implements the McCall's FCM model was unsuccessful. It is believed that the tool developed in this research is the first tool that implements the FCM structure to be developed locally.